H.E.B. Ecole Superieur d'Informatique

Laboratoire de C++ : projet 2

Starlight

Auteurs :
Paul Kriwin
39171@heb.be
Simon Placentino
39631@heb.be

Titulaire du cours :
Dr. Romain Absil
rabsil@heb.be

24 avril 2015

Table des matières

Chapitre 1

Introduction

`Starlight` est un petit jeu en deux dimensions se jouant sur une carte rectangulaire, comportant une source de lumière, émettant un rayon rectiligne. Le but du jeu est d'atteindre une cible avec ledit rayon, en évitant les obstacles via notamment des miroirs réfléchissant la lumière [1].

Ce document présente les différents détails de la réalisation du projet ; La section 2, décrit l'aspect formel du code (conventions de nommage, structure des fichier etc..).

Ensuite, la section 3 présente les classes utilisées et détaille leur contenu. La section 4 décrit la structure générale du programme qui est ensuite complétée par la section 5 décrivant les principaux algorithmes utilisés à travers le programme. Les sections suivantes décrivent respectivement les tests effectués et les points bonus réalisés. Enfin la section 9 conclut le document et est suivit par l'annexe A contenant les références.

1.1 Langage et compilation

Ce projet a été bâti à l'aide de **Qt Creator 5.4** [2], pour l'accès à la bibliothèque graphique, et compilé à l'aide de **g++** [3] dans sa version 4.8 (et compatible 4.9). La norme **ISO/IEC 14882 :2011** [4] , aussi appelée c++11, est celle utilisée. A cela s'ajoute certains flags [5] de compilation

g++ **-std=c++11 -Wextra -Wall -pedantic-errors**

- **-std=c++11** pour travailler en c++11,
- **-Wextra** qui nous permet d'avoir des messages d'avertissements supplémentaires (une classe non initialisée dans une classe dérivée, ...),

1. tiré des consignes du projet
2. voir Annexe A, QT Creator
3. voir Annexe A, GNU GCC
4. voir Annexe A, Catalogue des normes ISO
5. un flag est un argument de compilation optionnel

- **-Wall** qui nous permet d'ajouter de nouveaux messages d'avertissements (ordre de la liste d'initialisation, ...),
- **-pedantic-errors** qui transforme certains warnings [6] en erreurs.

`Ce projet est donc certifié "Warning Free"`

Pour des raisons pratiques, le projet est accompagné d'un fichier de type **Makefile** [7] qui s'occupe de

- nettoyer le répertoire à l'aide de la commande

  make_clean
- compiler l'ensemble du projet à l'aide de la commande

  make

6. un warning n'empêche pas une bonne compilation mais demande au programmeur de vérifier

7. voir Annexe A, GNU MakeFile

1.2 Documentation

L'ensemble du projet est documenté à l'aide des balises Qt [8]

```
/*!
* \brief Documentation de la methode enTete
* \param Description d'un parametre
* \return Ce qui est retourne
* \see Renvoie vers une autre documentation
*/
T enTete(R param);

/*!
* \brief Documentation de la variable
*/
T variable;
```

et cette documention[8] a est uniquement présente dans les fichiers **headers**. La documentation, au format LaTeXet html, a été généré à l'aide de l'outil Doxygen[8] dans sa dernière version disponible [9]. Les spécifications de compilations sont disponible dans le document "doxyConfig". La documentation U.M.L. [10] a été généré par l'intégration de l'outil **graphviz** [11]. Cette documentation est disponible dans les dossiers

`documentation/html/index.html`
`documentation/latex/documentation.pdf`

1.3 Contenu des fichiers

Chaque fichier contiendra les en-têtes, ou le code source, d'au maximum une seule classe et d'au maximum un seul namespace. Si la classe ne contient pas au moins une classe publique, le fichier portera le nom du namespace. Par exemple, le contenu du fichier hello.hpp peut être ainsi :

```
namespace unnamespace {class Hello {};}
```

```
namespace hello {}
```

```
class Hello {};
```

Le contenu de l'ensemble des fichiers est disponible dans le dossier

`src/`

8. voir Annexe A, Doxygen
9. 1.8.9.1
10. voir Annexe A, Unified Modeling Language
11. disponible dans les packages GNU/Linux - Ubuntu standards

1.4 Fichiers

1.4.1 noms

Le nom des fichiers est entièrement écrit en minuscule et porte le nom de la classe, ou du namespace qu'il contient.

headers

Les headers porteront l'extension **hpp**. Ils seront disposés dans le même dossier que leur fichier source **cpp** correspondant.

sources

Les fichiers sources porteront l'extension **cpp**. Ils seront disposé dans le même dossier que leur fichier header **hpp** correspondant.

test

Le nom des fichiers de test est composé du nom de la classe ou du namespace testé suivi de "test"

```
nuke.cpp → nuketest.cpp
mirror.cpp → mirrortest.cpp
```

L'ensemble de ces classes sont disponibles dans le dossier `test/`

1.4.2 html

Les documents `HTML` [12] disponible dans `ressources/other` sont des textes formatés à l'aide de `CSS` [13]. Ils sont utilisés lors de l'affichage des **règles** et du **logo**.

1.4.3 niveaux

Les fichiers de niveau qui peuvent être lu doivent être au format
- `.lvl` [14]
- `.mapl`

Ceux-ci sont composé de ligne significative [15] et structurés ainsi :
- taille du niveau,
- position de la source,
- position de la destination,
- élément du niveau,

12. voir Annexe A, HTML sur Wikipedia
13. voir Annexe A, CSS sur Wikipedia
14. Il est préférable d'utiliser celui-ci
15. c'est à dire que chaque ligne représente un objet à créer

– élément du niveau, ...

Les trois premiers sont donc nécessaire au bon fonctionnement d'une partie minimale.

1.5 Nom des namespace

Le nom d'un namespace sera exclusivement en minuscule.

```
namespace hebesi {}
```

1.6 Nom des classes

Les classes commencent toutes par une majuscule pour continuer, ensuite, en CamelCase [16]

```
class UneBonneClasse{};
class uneMauvaiseclasse{};
```

1.7 Nom des variables

Toutes les variables sont écrites en **camelCase** [17]

```
T uneBonneVariable;
T UneMauvaiseVariable;
T uneautreMauvaisevariable;
```

et possède des noms le plus explicite possible

```
T nb; // OK
T n{38.}; // NOK
T waveLength; // OK
```

Les variables de classes possèdent le même nom que le paramètre de constructeur qui l'initialisera. Le langage nous donne la possibilité de désambiguïser l'utilisation de ces variables à l'aide de

```
T var;
this->var;
```

et de la liste d'initialisation

```
UneClasse::UneClasse(T param) : param{param} {}
```

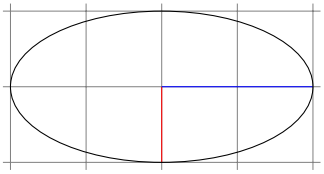
16. voir Annexe A, CamelCase sur Wikipedia
17. 16

Chapitre 2

Présentation succincte des classes

2.1 geometry

2.1.1 ellipse.hpp



Une ellipse [1] est un objet géométrique à deux dimensions représentée par une courbe plane fermée obtenu par découpe d'un cône sur un plan. Si ce dernier est perpendiculaire à l'axe du cône, l'ellipse sera alors un cercle. Éléments caractéristiques d'une ellipse :

- une coordonnée cartésienne, ou polaire, de son **centre** c,
- une distance séparant le centre de l'intersection avec la tangente parallèle à l'axe des ordonnées x_{radius},
- une distance séparant le centre de l'intersection avec la tangente parallèle à l'axe des abscisses y_{radius}.

Ces éléments nous permettront de tracer une ellipse selon l'équation :

$$\frac{(x - c_x)^2}{{x_{radius}}^2} + \frac{(y - c_y)^2}{{y_{radius}}^2} = 1$$

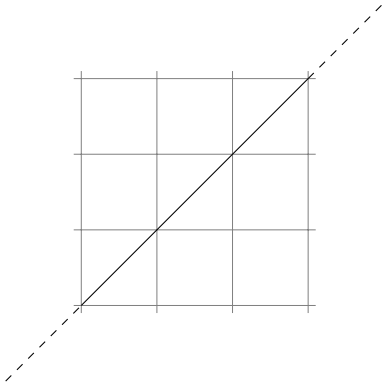
Cette classe peut tout à fait être instancié en objet géométrique elliptique et possède des méthodes d'interactions avec une droite [2].

1. voir Annexe A, Ellipse sur Wikipedia
2. L'algorithme est disponible dans la section du même nom ou dans la documentation accompagnant le projet

```
Point * Ellipse::getIntersectionsPoints(const Line &);
```

Dans le contexte présent, certains éléments du jeu seront des ellipse par le phénomène d'héritage mis en place dans le paradigme orienté objet de C++ [3]

2.1.2 line.hpp



Une `droite` [4] est une ligne sans épaisseur, rectiligne et infinie dans le plan. Pour exister, une droite aura besoin :

- d'un coefficient angulaire $m = \frac{\Delta y}{\Delta x}$ représentant la distance à parcourir sur l'axe des ordonnées pour une unité de distance sur l'axe des abscisses.
- d'un terme indépendant $p = \frac{y}{m \cdot x}$ représentant le décalage de chaque point sur l'axe des ordonnées,
- ou de deux points de coordonnées dans le plan a, b
- ou d'un point de coordonnées dans le plan a et d'un coefficient angulaire $m = \frac{\Delta y}{\Delta x}$.

Ces éléments nous permettent de tracer une droite selon l'équation :

$$y = m \cdot x + p$$

Il s'agit donc de cette dernière qui a été modélisé. Pour la gestion des droites verticales, un paramètre supplémentaire, et optionnel, a été rajouté au constructeur. Ainsi, une méthode de la classe utilities nous permet de savoir si le coefficient angulaire est dit "infini" et donc nous permettre de contenter la demande

```
bool * Line::isVertical();
```

3. `http://www.tutorialspoint.com/cplusplus/cpp_object_oriented.htm`
4. voir Annexe A, Droite sur Wikipedia

Ainsi, il est simple de savoir, en interieur comme en exterieur de la classe, si la droite est verticale ou non. [5] Il est aussi possible de savoir si un point fait parti de la droite courante.

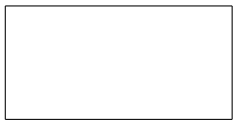
```
bool * Line::includes(const Point &) const;
```

Cette classe peut tout à fait être instancié en objet géométrique linéaire et possède des méthodes d'interactions avec d'autres droites [6].

```
Point * Line::getIntersectionPoint(const Line &);
```

Dans le contexte présent, certains éléments du jeu seront des droites par le phénomène d'héritage mis en place dans le paradigme orienté objet de C++ [7]

2.1.3 rectangle.hpp



Un rectangle [8] est une forme géométrique à 4 segments de droite [9] parallèle deux à deux.Ceux-ci vont donc former 4 angles droit ($\frac{\pi}{2}rad$) Cette forme peut être représentée par :

- la coordonnée du coin supérieur gauche $Sg = (x, y)$
- la grandeur des deux segments formant un angle de $\frac{\pi}{2}rad$ en ce point $hauteur$ et $largeur$.

Ainsi, il sera aisé de déterminer la position des autres coins

- $Sd = (Sg_x + largeur, Sg_y)$
- $Ig = (Sg_x, Sg_y + hauteur)$
- $Id = (Sg_x + largeur, Sg_y + hauteur)$

et de modéliser le rectangle à l'aide de 4 équations de droite, les objets Line. Cette classe peut tout à fait être instancié en objet géométrique rectangulaire et possède des méthodes d'interactions avec d'autres droites [10].

```
vector<Point> Rectangle::getIntersectionPoints(Line &);
```

5. cette méthode sera utile pour trouver l'intersection de deux droites puisque nous n'utilisons pas la forme $ax + bx + c = 0$

6. L'algorithme est disponible dans la section du même nom ou dans la documentation accompagnant le projet

7. `http://www.tutorialspoint.com/cplusplus/cpp_object_oriented.htm`

8. voir Annexe A, Rectangle sur Wikipedia

9. Un segment de droite est une partie de droite délimitée par deux points non confondus

10. L'algorithme est disponible dans la section du même nom ou dans la documentation accompagnant le projet

2.1.4 point.hpp

Un point [11] est un objet mathématique permettant de situer un élément dans un plan ou dans l'espace. Dans notre cas, plus spécifiquement dans un plan à deux dimensions. Celui-ci peut-être représenté de plusieurs manières dans le plan cartésienvoir Annexe A, Plan cartésien :

- sous la forme d'une coordonnées cartésienne à l'aide de
  - une origine,
    $$(0, 0)$$
  - deux vecteurs partant de cette origine et perpendiculaires,
    $$P = (\vec{x}, \vec{y})$$
- et sous la forme d'une coordonnée polaire à l'aide de
  - une origine,
    $$(0, 0)$$
  - une coordonnée radiale,
    $$r \in \mathbb{R}$$
  - une coordonnée angulaire,
    $$\alpha \in \mathbb{R}$$

2.1.5 utilities.hpp

Le namespace `utilities` mis en place ici est un ensemble de fonctions et valeurs constantes spécifiquement définies pour les calculs intervenant dans le projet.

constantes :

PI est un approximation de π sur 26 décimales,

PI_2 est une approximation de $\frac{\pi}{2}$ sur 26 décimales,

PI_4 est une approximation de $\frac{\pi}{4}$ sur 26 décimales,

EPSILON est une marge d'erreur de 10^{-7},

INF représente un nombre dit "infini" dans le milieu informatique.

fonctions :

Resolution d'équation du second degre

`utilitaire::secondDegreeEquationSolver`

11. voir Annexe A, Point sur Wikipedia

Transforme un angle exprime en radian en un angle exprime en degres

```
utilitaire::angleAsDegree
```

Permet de tester l egalite ou l inegalite entre deux nombre reels a un Epsilon d erreur

```
utilitaire::equals
utilitaire::greaterOrEquals
utilitaire::lessOrEquals
```

Permet de trouver le coefficiant angulaire a partir de deux points

```
utilitaire::slopeFromPoints
```

Permet de trouver la valeur tangante d'un angle en radian mais aussi de retourner une valeur particuliere pour la tangante de $\pi/2$

```
utilitaire::tan
```

Permet de savoir si α vaut $\frac{\pi}{2} + n * \pi \; n \in \mathbb{N}$

```
utilitaire::isHalfPiPlusNPi
```

Pour d'autres informations, consultez la documentation générée.

2.2 éléments

2.2.1 element.hpp

Cette classe, abstraite et donc non instanciable, représente un élément du jeu lié à un, et un seul, niveau du jeu. Cette classe permet donc d'établir une communication entre les différents éléments du niveau et le niveau lui-même à travers un référencement de ce dernier. Par le phénomène d'héritage, chaque élément se devra d'établir sa manière propre de réagir avec le niveau en lui exprimant :

- ses points d'intersection avec un rayon si il lui sont demandés,

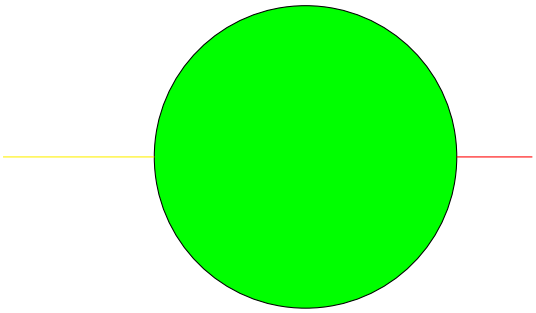
```
Element::includeRay(Ray);
```

- les actions à effectuer si un contact avec le rayon a eu lieu.

```
Element::reactToRay(Ray);
```

Il sera donc aisé pour le niveau de gérer les collisions des éléments de manière anonyme et optimale.

2.2.2 crystal.hpp



Cette classe est, par héritage, un élément ainsi qu'une ellipse. Il peut donc être modéliser graphiquement, entant que cette dernière, dans le plan et communique avec le niveau auquel il est lié.

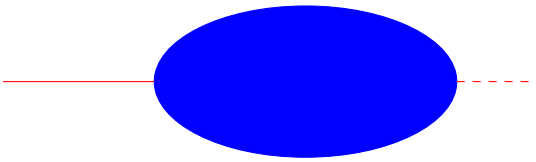
Le cristal modifie la longueur d'onde d'un rayon qui le traverse en lui augmentant ou en lui diminuant sa longueur d'onde. Il en devient nécessaire à la réussite d'une partie de jeu puisque les éléments `lens` requiert un rayon de longueur d'onde particulière. Puisque le rayon est visible il se doit de respecter le spectre lumineux visible $IR < wl < UV$ et donc un dépassement par le haut ou par le bas après amplification sera rétabli à la valeur logique la plus proche.

2.2.3 dest.hpp

Cette classe est, par héritage, un élément ainsi qu'un rectangle. Il peut donc être modéliser graphiquement, entant que cette dernière, dans le plan et communique avec le niveau auquel il est lié.

La destination est le but du jeu et l'atteindre terminera automatiquement la partie sur une victoire. La longueur d'onde du rayon n'a aucun importance.

2.2.4 lens.hpp



Cette classe est, par héritage, un élément ainsi qu'un rectangle. Il peut donc être modéliser graphiquement, entant que cette dernière, dans le plan et communique avec le niveau auquel il est lié. La lentille est un éventuel obstacle à un rayon puisqu'il ne le laissera passer que si la longueur d'onde de ce dernier respecte le critère

$$lens_{min} \leq ray_{\alpha} \leq lens_{max}$$

Deux issues sont alors possible :

- la longueur d'onde entre dans l'intervalle et le rayon est renouvelé après la lentille,
- elle ne rentre pas dans l'intervalle défini et le rayon sera terminé par la lentille

et ce même si le rayon tiré est tangent à la lentille. En effet, si la droite représentant le rayon est tangent à l'ellipse représentant la lentille, un point d'intersection existe entre les deux. Il est donc de ce point d'agir comme l'aurait fait l'entière lentille à tout autre point même si cette réaction particulière peut être discutable.

2.2.5 mirror.hpp

Cette classe est, par héritage, un élément ainsi qu'une droite. Le miroir peut donc être modéliser graphiquement, entant que cette dernière, dans le plan et communiquer avec le niveau auquel il est lié. Le miroir est, plus précisément, un segment de droite délimité par deux points. Puisque le miroir effectue une rotation autour d'un point $p \in mirroir$, il n'est pas directement délimité par ses deux extrémités mais bien par sa longueur, son angle, la position absolue (cartésienne) et relative (distance par rapport à une extrémité) de son point de rotation qui définissent, dynamiquement, ses points délimiteurs. Pour trouver les deux points il suffit de :

1. extrémité gauche = (x_{pivot} - position absolue, y_{pivot})
2. extrémité droite = (x_{gauche} + taille du segment, y_{pivot})
3. rotation des deux extrémité autour du point de pivot.

2.2.6 nuke.hpp

Cette classe est, par héritage, un élément ainsi qu'une ellipse. Il peut donc être modéliser graphiquement, entant que cette dernière, dans le plan et communique avec le niveau auquel il est lié. La bombe est une ellipse particulière puisqu'elle possède un $x_{radius} = y_{radius}$ lui donnant comme caractéristique d'être un cercle. Cet objet circulaire est l'objet à éviter lors d'une partie puisqu'il amène directement à la fin d'une partie.

2.2.7 ray.hpp

Cette classe est, par héritage, une droite. Ce rayon représente l'ensemble des points parcouru par un rayon laser et subit les interactions de son environnement comme dans la vie réelle :

- il sera reflété par un miroir selon le principe de réflexion $\alpha_i = \alpha_r$,
- il sera arrêté par les objets opaques (murs, source, destination).

Le rayon déclenche les réactions des éléments du jeu via la méthode `reactToRay` des héritiers d'Element.

2.2.8 source.hpp

Cette classe est, par héritage, un élément ainsi qu'un rectangle. Il peut donc être modéliser graphiquement, entant que cette dernière, dans le plan et communique avec le niveau auquel il est lié.

La source est le point d'émission du tout premier rayon du jeu selon un sens et une direction définis dans le fichier de niveau. Celui-ci ne peut être modifié directement en jeu. La source n'existe pas, c'est à dire qu'elle n'a aucun effet sur les rayons et les rayons n'ont aucun effet sur elle.

2.2.9 wall.hpp

Cette classe est, par héritage, un élément ainsi qu'une droite. Il peut donc être modéliser graphiquement, entant que cette dernière, dans le plan et communique avec le niveau auquel il est lié. Le mur est défini par deux points qui en font un segment de droite. Il est un objet fixe du jeu qui arrête un rayon et lui défini un point qui le transforme en segment de droite.

2.2.10 level.hpp

Le level est une classe qui est composée et qui s'occupe de gérer l'ensemble des éléments du jeu au niveau "business".

Celui-ci va donc recevoir les communications de ses éléments et agir en conséquence à l'aide de fonction récursive croisée

```
void computeRay(Ray);
void computeRays();
```

En effet, puisque la source créé un rayon dans le niveau, celui-ci va s'occuper de demander à chaque élément qui interagit avec le rayon précédent ce qui se passera ensuite et à partir de là, une réaction "boule de neige" se fera jusqu'à l'intersection d'un élément arrêtant le processus.

2.2.11 levelfactory.hpp

Le créateur de niveau est un `namespace` de méthodes qui va s'occuper de lire les données des éléments dans un fichier. Pour ce faire, il s'occupera de lire chaque type d'objet selon des règles différentes :

```
levelFactory::getSource;
levelFactory::getDestination;
levelFactory::getCrystal;
levelFactory::getLens;
levelFactory::getNuke;
levelFactory::getWall;
levelFactory::getMirror;
```

2.3 exception

2.3.1 starlightexception.hpp

Il est nécessaire, pour bon nombre des classes créées, de valider les arguments passés en paramètre dans le but de ne pas produire d'objets incohérents par rapport à l'analyse préalable du travail à fournir. Pour ce faire, des exceptions doivent être levées quand une instanciation créera un objet non désiré. Cette classe hérite de std : :exception appartenant à la librairie standard. Elle n'a aucune capacité supplémentaire mise à part être spécifique à ce projet.

Les différentes classes pouvant lever cette exception sont :

crystal si la taille de son rayon ne lui permet pas d'exister dans le plan,

lens si son intervalle de longueur d'onde n'est pas cohérent,

level si ses dimensions ne lui permettent pas d'exister dans le plan,

mirror si ses dimensions ne lui permettent pas d'exister dans le plan, si sa position ou son angle n'entre pas dans les limites imposées,

nuke si la taille de son rayon ne lui permet pas d'exister dans le plan,

ray si sa longueur d'onde n'entre pas dans l'intervalle cohérent imposé,

source si sa longueur d'onde n'entre pas dans l'intervalle cohérent imposé,

wall si ses points déterminants ne lui permettent pas d'exister dans le plan,

ellipse si ses dimensions ne lui permettent pas d'exister dans le plan,

rectangle si ses dimensions ne lui permettent pas d'exister dans le plan.

2.4 view

La vue du jeu est constituée de trois éléments ;
- `MainMenu` (widget),
- `LevelView` (widget),
- `la fenêtre principale` (MainWindow),

cette dernière s'occupe d'alterner l'affichage des deux premiers.

Ces éléments graphiques peuvent être classés en 3 catégories :

1. `éléments statiques` : N'étant pas directement manipulés par l'utilisateur, ces derniers représentent les éléments du niveau dont **l'affichage ne varie pas** au cours de la partie. Il s'agit des murs du niveaux (QGraphicsLineItems[15]), des lentilles, bombes, cristaux (QGraphicsEllipseItem[15]) et de la destination (QGraphicsRectItem[15]). Ces éléments sont construit au chargement du fichier de niveau grâce à des méthodes utilitaires contenues dans viewUtilities.hpp. Ils ne sont jamais mis à jours et sont détruits lorsque un nouveau niveau est chargé.

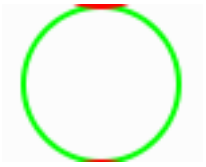


FIGURE 2.1 – Un cristal.

FIGURE 2.2 – Une destination.

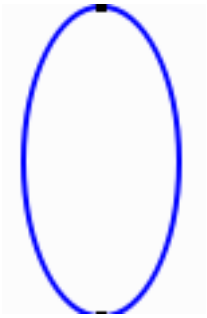


FIGURE 2.3 – Une lentille.

FIGURE 2.4 – Une bombe.

FIGURE 2.5 – Un mur.

2. `Semi dynamiques` : Il s'agit de l'affichage des rayons à partir de QGraphicsLineItems[15] dont la couleur varie selon la longueur d'onde du rayon représenté. Ils sont construits grâce à des méthodes utilitaires contenues dans viewUtilities.hpp et détruits à chaque modification de l'état du niveau.

FIGURE 2.6 – Un rayon changeant de couleur.

3. `dynamiques` : Ce sont les élément permettant de **recueillir les entrées utilisateur** et dont l'affichage évolue en fonction de l'état du niveau.

FIGURE 2.7 – La source éteinte.

FIGURE 2.8 – La source allumée.

2.4.1 sourceview.hpp

Cette classe représente la source sous l'état d'un `QGraphicsRectItem`, contenant un QRectF aux dimensions de la source. Il peut donc être utilisé dans un QGraphicsScene. Il est, donc, `lié à la source` et se met uniquement à jour lors d'une interaction de l'utilisateur (clic de souris). De plus, il interagit avec le modèle pour changer l'état de la source.

Il s'agit d'un objet héritant de QGraphicsRectItem[15] représentant la source lumineuse du niveau. Il permet lorsque l'utilisateur clique dessus, d'allumer/éteindre la source du niveau et ainsi provoquer la présence ou l'absence des rayons dans le niveau. [12]

2.4.2 mirrorview.hpp

Il s'agit d'un objet héritant de QGraphicsLineItem[15] représentant un miroir du niveau. Il permet une fois sélectionné en ayant cliqué dessus, de `faire dévier` les rayons l'atteignant en fonction des mouvements que l'utilisateur exercera sur lui : [13].

- touches pour le déplacer
- et flèches directionnelles pour le faire pivoter

FIGURE 2.9 – Un miroir.

FIGURE 2.10 – Un miroir et un rayon réfléchi.

12. *il émet un son et change sa couleur selon que l'on allume ou éteint la source.*
13. voir Annexe A, commandes du jeu

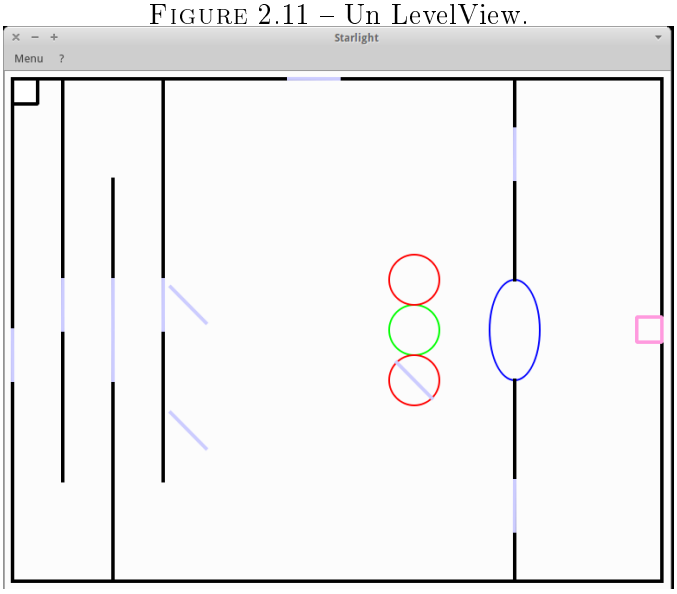
2.4.3 levelview.hpp

Il s'agit d'un **widget** hérité de QGraphicsView (utilisant, en son sein,le framework du même nom [14]), s'occupant de l'affichage du niveau, de la réception des entrées utilisateur et de l'interaction entre celles-ci et la partie métier.

Cette vue met en œuvre le patron de conception `Observateur Observé` initié par la partie métier (classe Level), de sorte qu'elle se contente seulement de mettre à jour l'affichage de l'état du niveau lorsqu'elle est notifiée de le faire par celui-ci.

Cette vue comporte une QGraphicScene [15] qui elle-même comporte des éléments graphiques (QGraphicItems[15]) représentant les différents éléments contenus dans le niveaux.

FIGURE 2.11 – Un LevelView.



2.4.4 mainmenu.hpp

Ce widget héritant de **QFrame** représente le menu principale du jeu. Il s'agit de la première fenêtre qui apparait à l'utilisateur lorsque l'on lance le programme.

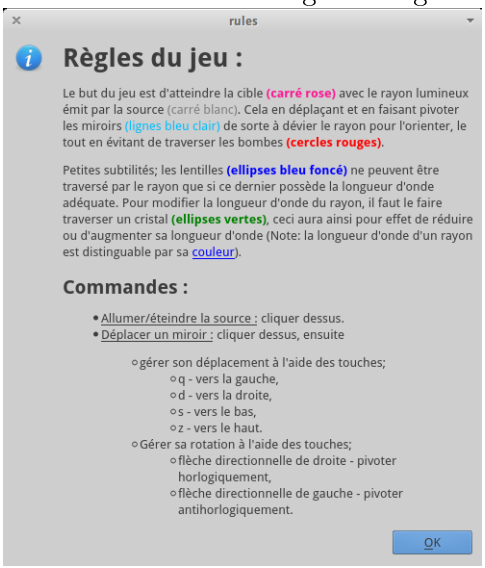
Cette fenêtre propose à travers de **QPushButtons**, diverses option telles que ;

- choisir un fichier de niveau (et ainsi démarrer une partie),
- consulter les règles du jeu (s'affichant dans une boîte de dialogue)
- ou quitter le jeu (arrête le programme).

14. voir Annexe A, Qt GraphicsView Framework
15. voir Annexe A, Qt QGraphicScene

FIGURE 2.12 – Un menu principal.

FIGURE 2.13 – Affichage des règles.



2.4.5 mainwindow.hpp

Il s'agit de la **fenêtre principale** du programme qui s'occupe d'alterner l'affichage du menu principal du jeu et l'affichage d'une partie. Elle permet également de proposer une barre d'outils durant l'affichage d'une partie permettant de

- recommencer le niveau en cours, revenir au menu principal,
- quitter le jeu ou connaitre les règles du jeu.

L'alternance de l'affichage est mis en œuvre grâce au mécanisme des signaux et `slots`[16] En effet MainWindow propose deux slots

MainWindow

```
public slots :

/* Permet d'afficher la vue du niveau. */
void MainWindow::displayLevel

/* Permet d'afficher le menu principal. */
void MainWindow::displayMainMenu
```

Ces slots sont par défaut connectés[17] aux signaux

LevelView

```
public signals :
void displayingStarted();
void displayingStopped();
```

Ainsi lorsque que l'utilisateur a sélectionné l'action de revenir au menu principal (à partir de la barre d'outils ou à partir de la boite de dialogue à la fin d'une partie),

1. la vue de niveau envoie le signal correspondant
2. cette dernière va ensuite **cacher la vue de niveau** et afficher le menu principal, (`displayingStopped()`) à la MainWindow auquel elle se rapporte,

Il en va de même pour afficher la vue du niveau,

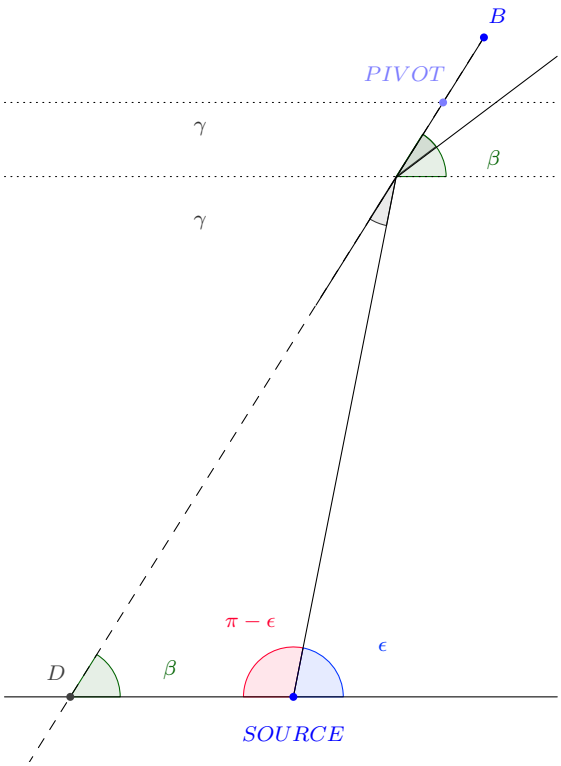
1. le signal newLevelFileSelected de MainMenu étant connecté au slot `setLevelFilePath()`),
2. ce dernier émet ensuite le signal `displayingStarted()`),
3. lorsque l'utilisateur choisi un fichier de niveau, la vue du menu principal va ainsi laisser place à la vue du niveau.

16. voir Annexe A, Signals and Slots
17. voir Annexe A, QObject : :connect dans Signals and slots

Chapitre 3

Détail des algorithmes utilisés

3.1 Algorithme de réflexion



α l'angle du nouveau rayon
β l'angle du miroir
γ l'angle incident
ϵ l'angle de la source

Par la somme des angles intérieurs d'un triangle valant πrad,

$$\gamma = (\pi - \beta - (\pi - \epsilon))$$

$\alpha = \beta - \gamma$ par les angles correspondants

3.2 Algorithme d'intersection

3.2.1 Intersection de deux droites

Soient

$$\begin{cases} y = m_1 \cdot x + p_1 \\ y = m_2 \cdot x + p_2 \end{cases}$$

deux droites sécantes ($m_1 \neq m_2$), résoudre l'équation

$$m_1 \cdot x + p_1 = m_2 \cdot x + p_2$$
$$m_1 \cdot x - m_2 \cdot x = p_2 - p_1$$
$$x \cdot (m_1 - m_2) = p_2 - p_1$$
$$x = \frac{(p_2 - p_1)}{(m_1 - m_2)}$$

nous donne l'abcisse du point d'intersection des deux droites. Il suffit ensuite de trouver y à l'aide d'une des deux équations.

3.2.2 Intersection d'une droite et d'un rectangle

Pour trouvez les intersections entre un rectangle et une droite, il suffit de chercher les intersections entre une droite et les quatres droites qui composent le rectangle (Cf. "Intersection de deux droites).

3.2.3 Intersection d'une droite et d'une ellipse

Soient

$$\begin{cases} y = m \cdot x + p \\ \frac{(x-c_x)^2}{{x_{radius}}^2} + \frac{(y-c_y)^2}{{y_{radius}}^2} = 1 \end{cases}$$

une droite et une ellipse quelconque, substituons la droite à l'équation de l'ellipse

$$\begin{cases} y = m \cdot x + p \\ \frac{(x-c_x)^2}{{x_{radius}}^2} + \frac{((m \cdot x+p)-c_y)^2}{{y_{radius}}^2} = 1 \end{cases}$$

$$\begin{cases} y = m \cdot x + p \\ \frac{(x-c_x)^2 * {y_{radius}}^2}{{x_{radius}}^2 * {y_{radius}}^2} + \frac{((m \cdot x+p)-c_y)^2 * {x_{radius}}^2}{{y_{radius}}^2 * {x_{radius}}^2} = 1 \end{cases}$$

3.3 Fonctionnement du moteur

Le moteur du jeu est articulé autour d'un mécanisme de **réactions en chaine** à l'exposition au rayon par les éléments, induite par l'allumage de la source et le mouvement des miroirs.

Lors de l'allumage de la source, celle-ci va appeler la méthode computeRays(sur la référence du niveau qu'elle contient en attribut. Cette méthode, la principale du jeu sert à calculer tous les rayons du niveau. Cette méthode appelle une autre méthode de Level, `ComputeRay(Ray)` qui elle, prenant en paramètre un rayon fraichement créé possédant un début une orientation mais pas encore de fin, va calculer cette dernière. C'est cette méthode qui va causer la "réaction en chaine " susmentionnée. En effet, cette méthode va à partir du rayon reçu en paramètre et de la méthode getEltsIntrajectory(Ray) (de level également), trouver le premier élément se trouvant sur la trajectoire du rayon et ainsi lui assigner sa position de fin avec le point d'intersection trouvé avec cet élément, le rayon sera ensuite stocké dans le vector prévu à cet effet en attribut de Level. Ensuite va être

Cette méthode pourra avoir plusieurs effet :

- créer une récursivité croisée en appelant la méthode computeRay(Ray) avec un nouveau rayon modifié créé à partir du paramètre de reactToray(Ray) créant ainsi de nouveaux rayons. C'est le cas de
  - `mirror`, qui modifie la direction du rayon,
  - `crystal`, qui modifie sa longueur d'onde
  - ou possiblement `lens` si la longueur d'onde du rayon correspond à ces bornes.
- Stopper net la progression du rayon, c'est le cas de wall, nuke, dest et lens si la longueur d'onde du rayon ne correspond pas à ces bornes.

Chapitre 4

Test effectués

4.1 Framework de test

Le Framework de test utilisé est "Catch"[??]. Ses avantages sont :
- un simple header est requis,
- **TEST_CASE** créant un bloc de tests,
- **SECTION** créant un sous bloc de tests,
- un ensemble de macros d'assertions sont disponibles :
  - **REQUIRE** qui attend une expression vraie,
  - **REQUIRE_FALSE** qui attend un expression fausse,
  - **REQUIRE_THROWS_AS** qui attend une erreur de type défini,
  - **REQUIRE_NO_THROW** qui n'attend pas d'erreur.

4.2 Tests unitaire

Chaque classe a été soumise à une batterie de tests unitaires. Pour se faire, chacune de ses méthodes s'est vu confirmé son bon fonctionnement au travers de cas dis "limites" et de cas dit "standards".

Les fichiers sources concernés se situe dans le dossier `test/` et peuvent éffectuer en décommentant la définition de #**RUN_TEST** du fichier **main.hpp** disponible à la racine du projet.

4.3 Bugs connus

Le programme ne contient aucuns bugs connus et ne présente aucune fuite mémoire en dehors de celles induite inévitablement par le framework d'interface graphique Qt.
Le modèle a été testé à l'aide de l'outil Valgrind [1].

1. voir Annexe A, Valgrind memory checker

Chapitre 5

Bonus

Ensemble des bonus terminés :

1. **les sons** [1] sont disponibles lors
   - de l'activation de la source,
   - de la désactivation de la source,
   - de l'activation de la destination,
   - ou de l'activation d'une bombe.

   Ils ont été ajouté au projet à l'aide du gestionnaires de ressource Qt [2] et activé à l'aide de la méthode statique [3]

   ```
   QSound::play(``:/prefix/alias'');
   ```

2. **les couleurs** [4] sont disponibles en fonction de la longueur d'onde du rayon
3. **l'icone du logiciel** disponible et a été désigné par **Kriwin Paul** sur un support libre d'image d'étoile.

1. voir Annexe A, Sound bible
2. voir Annexe A, Qt ressources system
3. voir Annexe A, QtMultimedia/QSound
4. voir Annexe A, calcul de longueur d'onde

Chapitre 6

Conclusion

Pour conclure, au travers de ce jeu l'algorithmique a pris une grande place. Si pour le simple utilisateur il parait simple et accessible, pour le développeur la mise en place a été rude. Du coté graphique, une grande facilité fut d'être aidé une très bonne documentation de la bibliothèque Qt standard qui se trouve être un modèle de rédaction de qualité.

Un énorme travail de recherche a du être fait au niveau géométrique ainsi qu'au niveau des spécifications du langage. Ce dernier a su, pour le binôme que nous avons formé durant plus d'un mois, se montrer à la hauteur et à se mettre en valeur par des attributs tout à fait intéressant (l'héritage multiple pour n'en citer qu'un).

Il en va de même pour la rédaction de ce document dans un format spécifique et compliqué (LaTeX)

Le travail à deux personnes est un enchainement de conflits et d'interactions intellectuelles musclées menant souvent à la refonte conjointe d'une idée personnelle.

Annexe A

Références

l'icone a été dessiné par **Kriwin Paul** sous `Gimp`
– `http://www.gimp.org/` consulté 24 avril 2015,

L'ensemble des sons disponibles dans `ressources/sounds` proviennent de `Sound bible` sous la license `Attribution 3.0 License` :
– `http://soundbible.com`, consulté 24 avril 2015
– `https://creativecommons.org/licenses/by/3.0/` consulté 24 avril 2015,

calcul de la longueur d'onde :
– `http://www.physics.sfasu.edu/astro/color/spectra.html` consulté 24 avril 2015,

`Catch Framework`
– `https://github.com/philsquared/Catch` consulté 24 avril 2015,

`U.M.L. - Unified Modeling Language`
– `http://www.uml.org/` consulté 24 avril 2015,

`Doxygen`
– `http://doxygen.org/` consulté 24 avril 2015,

`Plan cartésien`
– `http://www.cslaval.qc.ca/sitsatlll/maths2003/cartesien.html` consulté 24 avril 2015,

`Qt Creator` disponible dans sa dernière version sur
– `http://www.qt.io/developers/` consulté le 24 avril 2015,

`Catalogue des normes ISO`

– http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372 consulté le 24 avril 2015,

GNU GCC
– https://gcc.gnu.org/ consulté le 24 avril 2015,

GNU MakeFile
– https://www.gnu.org/software/make/manual/html_node/Makefiles.html consulté le 24 avril 2015,

HTML
– https://fr.wikipedia.org/wiki/Hypertext_Markup_Language consulté le 24 avril 2015,

CSS
– https://fr.wikipedia.org/wiki/Feuilles_de_style_en_cascade consulté le 24 avril 2015,

CamelCase
– https://fr.wikipedia.org/wiki/CamelCase consulté le 24 avril 2015,

Ellipse
– https://fr.wikipedia.org/wiki/Ellipse_%28math%C3%A9matiques%29 consulté le 24 avril 2015,

Droite
– https://fr.wikipedia.org/wiki/Droite_%28math%C3%A9matiques%29 consulté le 24 avril 2015,

Point
– https://fr.wikipedia.org/wiki/Point_%28g%C3%A9om%C3%A9trie%29 consulté le 24 avril 2015,

Rectangle
– https://fr.wikipedia.org/wiki/Rectangle consulté le 24 avril 2015,

Qt Ressource System
– http://doc.qt.io/qt-5/resources.html consulté le 24 avril 2015,

QSound
– http://doc.qt.digia.com/qt-maemo/qsound.html consulté le 24 avril 2015,

GraphicsView
– http://doc.qt.io/qt-4.8/graphicsview.html consulté le 24 avril 2015,

GraphicsView Framework
– http://doc.qt.io/qt-4.8/graphicsview.html#classes-in-the-graphics-view-framew consulté le 24 avril 2015,

Signals and slots
– http://doc.qt.io/qt-4.8/signalsandslots.html consulté le 24 avril 2015,

texttt Commandes et règles
– ressources/other/rules.html,

texttt Valgrind memory checker
– http://valgrind.org/, consulté le 24 avril 2015,